GALAHAD BGO

USER DOCUMENTATION GALAHAD Optimization Library version 5.1

1 SUMMARY

This package uses a **multi-start trust-region method to find an approximation to the global minimizer of a differentiable objective function $f(\mathbf{x})$ of n variables x, subject to simple bounds $\mathbf{x}^l \leq \mathbf{x} \leq \mathbf{x}^u$ on the variables.** Here, any of the components of the vectors of bounds $\mathbf{x}^l$ and $\mathbf{x}^u$ may be infinite. The method offers the choice of direct and iterative solution of the key trust-region subproblems, and is suitable for large problems. First derivatives are required, and if second derivatives can be calculated, they will be exploited—if the product of second derivatives with a vector may be found but not the derivatives themselves, that may also be exploited.

The package offers both random multi-start and local-minimize-and probe methods to try to locate the global minimizer. There are no theoretical guarantees unless the sampling is huge, and realistically the success of the methods decreases as the dimension and nonconvexity increase.

ATTRIBUTES — Versions: `GALAHAD_BGO_single`, `GALAHAD_BGO_double`. **Uses:** `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_NLPT`, `GALAHAD_USERDATA`, `GALAHAD_SPECFILE`, `GALAHAD_SPACE`, `GALAHAD_RAND`, `GALAHAD_NORMS`, `GALAHAD_UGO`, `GALAHAD_LHS` and `GALAHAD_TRB`. **Date:** July 2016. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003.

2 HOW TO USE THE PACKAGE

The package is available with single, double and (if available) quadruple precision reals, and either 32-bit or 64-bit integers. Access to the 32-bit integer, single precision version requires the `USE` statement

```
USE GALAHAD_BGO_single
```

with the obvious substitution `GALAHAD_BGO_double`, `GALAHAD_BGO_quadruple`, `GALAHAD_BGO_single_64`, `GALAHAD_BGO_double_64` and `GALAHAD_BGO_quadruple_64` for the other variants.

If it is required to use more than one of the modules at the same time, the derived types `SMT_type`, `GALAHAD_userdata_type`, `BGO_time_type`, `BGO_control_type`, `BGO_inform_type`, `BGO_data_type` and `NLPT_problem_type`, (Section 2.3) and the subroutines `BGO_initialize`, `BGO_solve`, `BGO_terminate`, (Section 2.4) and `BGO_read_specfile` (Section 2.8) must be renamed on one of the `USE` statements.

2.1 Matrix storage formats

If available, the Hessian matrix $\mathbf{H} = \nabla_{xx} f(\mathbf{x})$ may be stored in a variety of input formats.

2.1.1 Dense storage format

The matrix $\mathbf{H}$ is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. Since $\mathbf{H}$ is symmetric, only the lower triangular part (that is the part h_{ij} for $1 \leq j \leq i \leq n$) need be held. In this case the lower triangle should be stored by rows, that is component $i*(i-1)/2+j$ of the storage array `H%val` will hold the value h_{ij} (and, by symmetry, h_{ji}) for $1 \leq j \leq i \leq n$.

2.1.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l-th entry, $1 \leq l \leq$ H%ne, of $\mathbf{H}$, its row index i, column index j and value h_{ij}, $1 \leq j \leq i \leq n$, are stored in the l-th components of the integer arrays H%row, H%col and real array H%val, respectively. Note that only the entries in the lower triangle should be stored.

2.1.3 Sparse row-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in row i appear directly before those in row $i+1$. For the i-th row of $\mathbf{H}$, the i-th component of the integer array H%ptr holds the position of the first entry in this row, while H%ptr $(n+1)$ holds the total number of entries plus one. The column indices j, $1 \leq j \leq i$, and values h_{ij} of the entries in the i-th row are stored in components $l =$ H%ptr(i), ...,H%ptr $(i+1)-1$ of the integer array H%col, and real array H%val, respectively. Note that as before only the entries in the lower triangle should be stored. For sparse matrices, this scheme almost always requires less storage than its predecessor.

2.1.4 Diagonal storage format

If $\mathbf{H}$ is diagonal (i.e., $h_{ij} = 0$ for all $1 \leq i \neq j \leq n$) only the diagonals entries h_{ii}, $1 \leq i \leq n$, need be stored, and the first n components of the array H%val may be used for the purpose.

2.2 Real and integer kinds

We use the terms integer and real to refer to the fortran keywords REAL(rp_) and INTEGER(ip_), where rp_ and ip_ are the relevant kind values for the real and integer types employed by the particular module in use. The former are equivalent to default REAL for the single precision versions, DOUBLE PRECISION for the double precision cases and quadruple-precision if 128-bit reals are available, and correspond to rp_ = real32, rp_ = real64 and rp_ = real128 respectively as defined by the fortran iso_fortran_env module. The latter are default (32-bit) and long (64-bit) integers, and correspond to ip_ = int32 and ip_ = int64, respectively, again from the iso_fortran_env module.

2.3 The derived data types

Seven derived data types are accessible from the package.

2.3.1 The derived data type for holding matrices

The derived data type SMT_TYPE is used to hold the Hessian matrix $\mathbf{H}$ if this is available. The components of SMT_TYPE used here are:

n is a scalar component of type INTEGER(ip_), that holds the dimension of the matrix.

ne is a scalar variable of type INTEGER(ip_), that holds the number of matrix entries.

type is a rank-one allocatable array of type default CHARACTER, that is used to indicate the matrix storage scheme used. Its precise length and content depends on the type of matrix to be stored (see §2.3.2).

val is a rank-one allocatable array of type REAL(rp_) and dimension at least ne, that holds the values of the entries. Each pair of off-diagonal entries $h_{ij} = h_{ji}$ of the *symmetric* matrix $\mathbf{H}$ is represented as a single entry (see §2.1.1–2.1.3). Any duplicated entries that appear in the sparse co-ordinate or row-wise schemes will be summed.

row is a rank-one allocatable array of type INTEGER(ip_), and dimension at least ne, that may hold the row indices of the entries. (see §2.1.2).

col is a rank-one allocatable array of type INTEGER(ip_), and dimension at least ne, that may hold the column indices of the entries (see §2.1.2–2.1.3).

ptr is a rank-one allocatable array of type INTEGER(ip_), and dimension at least n + 1, that may hold the pointers to the first entry in each row (see §2.1.3).

2.3.2 The derived data type for holding the problem

The derived data type NLPT_problem_type is used to hold the problem. The relevant components of NLPT_problem_type are:

n is a scalar variable of type INTEGER(ip_), that holds the number of optimization variables, n.

H is scalar variable of type SMT_TYPE that holds the Hessian matrix **H**. The following components are used here:

H%type is an allocatable array of rank one and type default CHARACTER, that is used to indicate the storage scheme used. If the dense storage scheme (see Section 2.1.1) is used, the first five components of H%type must contain the string DENSE. For the sparse co-ordinate scheme (see Section 2.1.2), the first ten components of H%type must contain the string COORDINATE, for the sparse row-wise storage scheme (see Section 2.1.3), the first fourteen components of H%type must contain the string SPARSE_BY_ROWS, and for the diagonal storage scheme (see Section 2.1.4), the first eight components of H%type must contain the string DIAGONAL.

For convenience, the procedure SMT_put may be used to allocate sufficient space and insert the required keyword into H%type. For example, if nlp is of derived type BGO_problem_type and involves a Hessian we wish to store using the co-ordinate scheme, we may simply

```
CALL SMT_put( nlp%H%type, 'COORDINATE' )
```

See the documentation for the GALAHAD package SMT for further details on the use of SMT_put.

H%ne is a scalar variable of type INTEGER(ip_), that holds the number of entries in the **lower triangular** part of **H** in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be set for any of the other three schemes.

H%val is a rank-one allocatable array of type REAL(rp_), that holds the values of the entries of the **lower triangular** part of the Hessian matrix **H** in any of the storage schemes discussed in Section 2.1.

H%row is a rank-one allocatable array of type INTEGER(ip_), that holds the row indices of the **lower triangular** part of **H** in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be allocated for any of the other three schemes.

H%col is a rank-one allocatable array variable of type INTEGER(ip_), that holds the column indices of the **lower triangular** part of **H** in either the sparse co-ordinate (see Section 2.1.2), or the sparse row-wise (see Section 2.1.3) storage scheme. It need not be allocated when the dense or diagonal storage schemes are used.

H%ptr is a rank-one allocatable array of dimension n+1 and type INTEGER(ip_), that holds the starting position of each row of the **lower triangular** part of **H**, as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.1.3). It need not be allocated when the other schemes are used.

G is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the gradient **g** of the objective function. The j-th component of G, $j = 1,\ldots,n$, contains $\mathbf{g}_j$. These are equivalently the values **z** of estimates of the dual variables corresponding to the simple bound constraints (see Section 4).

f is a scalar variable of type REAL(rp_), that holds the value of the objective function.

X_l is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the vector of lower bounds $\mathbf{x}^l$ on the the variables. The j-th component of X_l, $j = 1,\ldots,n$, contains $\mathbf{x}^l_j$. Infinite bounds are allowed by setting the corresponding components of X_l to any value smaller than -infinity, where infinity is a component of the control array control (see Section 2.3.3).

X_u is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the vector of upper bounds $\mathbf{x}^u$ on the variables. The j-th component of X_u, $j = 1,\ldots,n$, contains $\mathbf{x}^u_j$. Infinite bounds are allowed by setting the corresponding components of X_u to any value larger than that infinity, where infinity is a component of the control array control (see Section 2.3.3).

X is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the values $\mathbf{x}$ of the optimization variables. The j-th component of X, $j = 1,\ldots,n$, contains x_j.

pname is a scalar variable of type default CHARACTER and length 10, which contains the "name" of the problem for printing. The default "empty" string is provided.

VNAMES is a rank-one allocatable array of dimension n and type default CHARACTER and length 10, whose j-th entry contains the "name" of the j-th variable for printing. This is only used if "debug"printing control%print_level > 4) is requested, and will be ignored if the array is not allocated.

2.3.3 The derived data type for holding control parameters

The derived data type BGO_control_type is used to hold controlling data. Default values may be obtained by calling BGO_initialize (see Section 2.4.1), while components may also be changed by calling GALAHAD_BGO_read_spec (see Section 2.8.1). The components of BGO_control_type are:

error is a scalar variable of type INTEGER(ip_), that holds the stream number for error messages. Printing of error messages in BGO_solve and BGO_terminate is suppressed if error ≤ 0. The default is error = 6.

out is a scalar variable of type INTEGER(ip_), that holds the stream number for informational messages. Printing of informational messages in BGO_solve is suppressed if out < 0. The default is out = 6.

print_level is a scalar variable of type INTEGER(ip_), that is used to control the amount of informational output which is required. No informational output will occur if print_level ≤ 0. If print_level $= 1$, a single line of output will be produced for each iteration of the process. If print_level ≥ 2, this output will be increased to provide significant detail of each iteration. The default is print_level = 0.

attempts_max is a scalar variable of type INTEGER(ip_), that specifies the maximum number of random searches from the best point found so far. The default is attempts_max = 10.

max_evals is a scalar variable of type INTEGER(ip_), that gives the maximum number of function evaluations that are allowed. The default is max_evals = 10000.

sampling_strategy is a scalar variable of type INTEGER(ip_), that is used to specify the sampling strategy used. Possible values are

1. Sample uniformly from the feasible region
2. Sample from the midpoint of a Latin hypercube
3. Sample uniformly within a Latin hypercube

The default is sampling_strategy = 1.

hypercube_discretization is a scalar variable of type INTEGER(ip_), that specifies the Latin hypercube discretization in each dimension (for sampling stategies sampling_strategy = 2 and 3). The default is hypercube_discretization = 2.

alive_unit is a scalar variable of type INTEGER(ip_). If alive_unit > 0, a temporary file named alive_file (see below) will be created on stream number alive_unit on initial entry to GALAHAD_BGO_solve, and execution of GALAHAD_BGO_solve will continue so long as this file continues to exist. Thus, a user may terminate execution simply by removing the temporary file from this unit. If alive_unit ≤ 0, no temporary file will be created, and execution cannot be terminated in this way. The default is alive_unit $= 60$.

infinity is a scalar variable of type REAL(rp_), that is used to specify which constraint bounds are infinite. Any bound larger than infinity in modulus will be regarded as infinite. The default is infinity $= 10^{19}$.

obj_unbounded is a scalar variable of type default REAL(rp_), that specifies smallest value of the objective function that will be tolerated before the problem is declared to be unbounded from below. The default is potential_unbounded $= -u^{-2}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_BGO_double).

cpu_time_limit is a scalar variable of type REAL(rp_), that is used to specify the maximum permitted CPU time. Any negative value indicates no limit will be imposed. The default is cpu_time_limit = - 1.0.

clock_time_limit is a scalar variable of type REAL(rp_), that is used to specify the maximum permitted elapsed system clock time. Any negative value indicates no limit will be imposed. The default is clock_time_limit = - 1.0.

hessian_available is a scalar variable of type default LOGICAL, that should be set .TRUE. if the user will provide second derivatives (either by providing an appropriate evaluation routine to the solver or by reverse communication, see Section 2.6), and .FALSE. if the second derivatives are not explicitly available. The default is hessian_available = .TRUE..

random_multistart is a scalar variable of type default LOGICAL, that should be set .TRUE. if a random-multistart method is to be used or .FALSE. if a local-minimize-and-probe approach is preferred . The default is random_multistart = .FALSE..

space_critical is a scalar variable of type default LOGICAL, that must be set .TRUE. if space is critical when allocating arrays and .FALSE. otherwise. The package may run faster if space_critical is .FALSE. but at the possible expense of a larger storage requirement. The default is space_critical = .FALSE..

deallocate_error_fatal is a scalar variable of type default LOGICAL, that must be set .TRUE. if the user wishes to terminate execution if a deallocation fails, and .FALSE. if an attempt to continue will be made. The default is deallocate_error_fatal = .FALSE..

alive_file is a scalar variable of type default CHARACTER and length 30, that gives the name of the temporary file whose removal from stream number alive_unit terminates execution of GALAHAD_BGO_solve. The default is alive_unit = ALIVE.d.

prefix is a scalar variable of type default CHARACTER and length 30, that may be used to provide a user-selected character string to preface every line of printed output. Specifically, each line of output will be prefaced by the string prefix(2:LEN(TRIM(prefix))-1), thus ignoreing the first and last non-null components of the supplied string. If the user does not want to preface lines by such a string, they may use the default prefix = "".

TRB_control is a scalar variable of type TRB_control_type whose components are used to control the local multivariate optimization aspects of the calculation, as performed by the package GALAHAD_TRB. See the specification sheet for the package GALAHAD_TRB for details, and appropriate default values.

UGO_control is a scalar variable of type UGO_control_type whose components are used to control the univariate global optimization calculation (if any), performed by the package GALAHAD_UGO. See the specification sheet for the package GALAHAD_UGO for details, and appropriate default values.

LHS_control is a scalar variable of type LHS_control_type whose components are used to control the Latox hypercube sampling algorithm as performed by the package GALAHAD_LHS. See the specification sheet for the package GALAHAD_LHS for details, and appropriate default values.

2.3.4 The derived data type for holding timing information

The derived data type BGO_time_type is used to hold elapsed CPU and system clock times for the various parts of the calculation. The components of BGO_time_type are:

total is a scalar variable of type default REAL, that gives the CPU total time spent in the package.

univariate_global is a scalar variable of type REAL(rp_), that gives the CPU time spent performing univariate global optimization.

multivariate_local is a scalar variable of type REAL(rp_), that gives the CPU time spent performing multivariate local optimization.

clock_total is a scalar variable of type default REAL, that gives the total elapsed system clock time spent in the package.

clock_univariate_global is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent performing univariate global optimization.

clock_multivariate_local is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent performing multivariate local optimization.

2.3.5 The derived data type for holding informational parameters

The derived data type BGO_inform_type is used to hold parameters that give information about the progress and needs of the algorithm. The components of BGO_inform_type are:

status is a scalar variable of type INTEGER(ip_), that gives the exit status of the algorithm. See Sections 2.6 and 2.7 for details.

alloc_status is a scalar variable of type INTEGER(ip_), that gives the status of the last attempted array allocation or deallocation. This will be 0 if status = 0.

bad_alloc is a scalar variable of type default CHARACTER and length 80, that gives the name of the last internal array for which there were allocation or deallocation errors. This will be the null string if status = 0.

f_eval is a scalar variable of type INTEGER(ip_), that gives the total number of objective function evaluations performed.

g_eval is a scalar variable of type INTEGER(ip_), that gives the total number of objective function gradient evaluations performed.

h_eval is a scalar variable of type INTEGER(ip_), that gives the total number of objective function Hessian evaluations performed.

obj is a scalar variable of type REAL(rp_), that holds the value of the objective function at the best estimate of the solution found.

norm_pg is a scalar variable of type REAL(rp_), that holds the value of the norm of the projected gradient of the objective function at the best estimate of the solution found.

time is a scalar variable of type BGO_time_type whose components are used to hold elapsed elapsed CPU and system clock times for the various parts of the calculation (see Section 2.3.4).

TRB_inform is a scalar variable of type TRB_inform_type whose components give information about the progress and needs of the local multivariate optimization stages of the algorithm performed by the package GALAHAD_TRB. See the specification sheet for the package GALAHAD_TRB for details.

UGO_inform is a scalar variable of type UGO_inform_type whose components give information about the progress and needs of the univariate global optimization stages of the algorithm performed by the package GALAHAD_UGO. See the specification sheet for the package GALAHAD_UGO for details.

LHS_inform is a scalar variable of type LHS_inform_type whose components give information about the progress and needs of the Latix hypercube sampling algorithm as performed by the package GALAHAD_LHS. See the specification sheet for the package GALAHAD_LHS for details.

2.3.6 The derived data type for holding problem data

The derived data type BGO_data_type is used to hold all the data for a particular problem, or sequences of problems with the same structure, between calls of BGO procedures. This data should be preserved, untouched (except as directed on return from GALAHAD_BGO_solve with positive values of inform%status, see Section 2.6), from the initial call to BGO_initialize to the final call to BGO_terminate.

2.3.7 The derived data type for holding user data

The derived data type GALAHAD_userdata_type is available from the package GALAHAD_userdata to allow the user to pass data to and from user-supplied subroutines for function and derivative calculations (see Section 2.5). Components of variables of type GALAHAD_userdata_type may be allocated as necessary. The following components are available:

integer is a rank-one allocatable array of type INTEGER(ip_).

real is a rank-one allocatable array of type default REAL(rp_)

complex is a rank-one allocatable array of type default COMPLEX (double precision complex in GALAHAD_BGO_double).

character is a rank-one allocatable array of type default CHARACTER.

logical is a rank-one allocatable array of type default LOGICAL.

integer_pointer is a rank-one pointer array of type INTEGER(ip_).

real_pointer is a rank-one pointer array of type default REAL(rp_)

complex_pointer is a rank-one pointer array of type default COMPLEX (double precision complex in GALAHAD_BGO_double).

character_pointer is a rank-one pointer array of type default CHARACTER.

logical_pointer is a rank-one pointer array of type default LOGICAL.

2.4 Argument lists and calling sequences

There are three procedures for user calls (see Section 2.8 for further features):

1. The subroutine BGO_initialize is used to set default values, and initialize private data, before solving one or more problems with the same sparsity and bound structure.
2. The subroutine BGO_solve is called to solve the problem.
3. The subroutine BGO_terminate is provided to allow the user to automatically deallocate array components of the private data, allocated by BGO_solve, at the end of the solution process. It is important to do this if the data object is re-used for another problem **with a different structure** since BGO_initialize cannot test for this situation, and any existing associated targets will subsequently become unreachable.

We use square brackets [ ] to indicate OPTIONAL arguments.

2.4.1 The initialization subroutine

Default values are provided as follows:

```
CALL BGO_initialize( data, control, inform )
```

data is a scalar INTENT(INOUT) argument of type BGO_data_type (see Section 2.3.6). It is used to hold data about the problem being solved.

control is a scalar INTENT(OUT) argument of type BGO_control_type (see Section 2.3.3). On exit, control contains default values for the components as described in Section 2.3.3. These values should only be changed after calling BGO_initialize.

inform is a scalar INTENT(OUT) argument of type BGO_inform_type (see Section 2.3.5). A successful call to BGO_initialize is indicated when the component status has the value 0. For other return values of status, see Section 2.7.

2.4.2 The minimization subroutine

The minimization algorithm is called as follows:

```
CALL BGO_solve( nlp, control, inform, data, userdata[, eval_F, eval_G,       &
                eval_H, eval_HPROD, eval_SHPROD, eval_PREC] )
```

nlp is a scalar INTENT(INOUT) argument of type NLPT_problem_type (see Section 2.3.2). It is used to hold data about the problem being solved. For a new problem, the user must allocate all the array components, and set values for nlp%n and the required integer components of nlp%H if second derivatives will be used. Users are free to choose whichever of the matrix formats described in Section 2.1 is appropriate for $\mathbf{H}$ for their application.

The component nlp%X must be set to an initial estimate, $\mathbf{x}^0$, of the minimization variables. A good choice will increase the speed of the package, but the underlying method is designed to converge (at least to a local solution) from an arbitrary initial guess.

On exit, the component nlp%X will contain the best estimates of the minimization variables $\mathbf{x}$, while nlp%G will contain the best estimates of the dual variables $\mathbf{z}$.

Restrictions: nlp%n $>$ 0 and nlp%H%type $\in$ {'DENSE','COORDINATE','SPARSE_BY_ROWS','DIAGONAL'}.

control is a scalar INTENT(IN) argument of type BGO_control_type (see Section 2.3.3). Default values may be assigned by calling BGO_initialize prior to the first call to BGO_solve.

inform is a scalar INTENT(INOUT) argument of type BGO_inform_type (see Section 2.3.5). On initial entry, the component status must be set to the value 1. Other entries need note be set. A successful call to BGO_solve is indicated when the component status has the value 0. For other return values of status, see Sections 2.6 and 2.7.

data is a scalar INTENT(INOUT) argument of type BGO_data_type (see Section 2.3.6). It is used to hold data about the problem being solved. With the possible exceptions of the components eval_status and U (see Section 2.6), it must not have been altered **by the user** since the last call to BGO_initialize.

userdata is a scalar INTENT(INOUT) argument of type GALAHAD_userdata_type whose components may be used to communicate user-supplied data to and from the OPTIONAL subroutines eval_F, eval_G, eval_H and eval_HPROD (see Section 2.3.7).

eval_F is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the objective function $f(\mathbf{x})$ at a given vector $\mathbf{x}$. See Section 2.5.1 for details. If eval_F is present, it must be declared EXTERNAL in the calling program. If eval_F is absent, GALAHAD_BGO_solve will use reverse communication to obtain objective function values (see Section 2.6).

eval_G is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the gradient of the objective function $\nabla_x f(\mathbf{x})$ at a given vector $\mathbf{x}$. See Section 2.5.2 for details. If eval_G is present, it must be declared EXTERNAL in the calling program. If eval_G is absent, GALAHAD_BGO_solve will use reverse communication to obtain gradient values (see Section 2.6).

eval_H is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the Hessian of the objective function $\nabla_{xx} f(\mathbf{x})$ at a given vector $\mathbf{x}$. See Section 2.5.3 for details. If eval_H is present, it must be declared EXTERNAL in the calling program. If eval_H is absent, GALAHAD_BGO_solve will use reverse communication to obtain Hessian function values (see Section 2.6).

eval_HPROD is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the product $\nabla_{xx} f(\mathbf{x})\mathbf{v}$ of the Hessian of the objective function $\nabla_{xx} f(\mathbf{x})$ with a given vector $\mathbf{v}$. See Section 2.5.4 for details. If eval_HPROD is present, it must be declared EXTERNAL in the calling program. If eval_HPROD is absent, GALAHAD_BGO_solve will use reverse communication to obtain Hessian-vector products (see Section 2.6).

eval_SHPROD is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the product $\nabla_{xx} f(\mathbf{x})\mathbf{v}$ of the Hessian of the objective function $\mathbf{u} = \nabla_{xx} f(\mathbf{x})$ with a given *sparse* vector $\mathbf{v}$, and to return the nonzero components of the resulting $\mathbf{u}$. See Section 2.5.5 for details. If eval_SHPROD is present, it must be declared EXTERNAL in the calling program. If eval_SHPROD is absent, GALAHAD_BGO_solve will use reverse communication to obtain Hessian-sparse-vector products (see Section 2.6).

eval_PREC is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the product $\mathbf{P}(\mathbf{x})\mathbf{v}$ of the user's preconditioner with a given vector $\mathbf{v}$. See Section 2.5.6 for details. If eval_PREC is present, it must be declared EXTERNAL in the calling program. If eval_PREC is absent, GALAHAD_BGO_solve will use reverse communication to obtain products with the preconditioner (see Section 2.6).

2.4.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

```
CALL BGO_terminate( data, control, inform )
```

data is a scalar INTENT(INOUT) argument of type BGO_data_type exactly as for BGO_solve, which must not have been altered **by the user** since the last call to BGO_initialize. On exit, array components will have been deallocated.

control is a scalar INTENT(IN) argument of type BGO_control_type exactly as for BGO_solve.

inform is a scalar INTENT(OUT) argument of type BGO_inform_type exactly as for BGO_solve. Only the component status will be set on exit, and a successful call to BGO_terminate is indicated when this component status has the value 0. For other return values of status, see Section 2.7.

2.5 Function and derivative values

2.5.1 The objective function value via internal evaluation

If the argument eval_F is present when calling GALAHAD_BGO_solve, the user is expected to provide a subroutine of that name to evaluate the value of the objective function $f(\mathbf{x})$. The routine must be specified as

```
SUBROUTINE eval_F( status, X, userdata, f )
```

whose arguments are as follows:

status is a scalar INTENT(OUT) argument of type INTEGER(ip_), that should be set to 0 if the routine has been able to evaluate the objective function and to a non-zero value if the evaluation has not been possible.

X is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{x}$.

userdata is a scalar INTENT(INOUT) argument of type GALAHAD_userdata_type whose components may be used to communicate user-supplied data to and from the subroutines eval_F, eval_G, eval_H, eval_HPROD and eval_PREC (see Section 2.3.7).

f is a scalar INTENT(OUT) argument of type REAL(rp_), that should be set to the value of the objective function $f(\mathbf{x})$ evaluated at the vector $\mathbf{x}$ input in X.

2.5.2 Gradient values via internal evaluation

If the argument eval_G is present when calling GALAHAD_BGO_solve, the user is expected to provide a subroutine of that name to evaluate the value of the gradient the objective function $\nabla_x f(\mathbf{x})$. The routine must be specified as

```
SUBROUTINE eval_G( status, X, userdata, G )
```

whose arguments are as follows:

status is a scalar INTENT(OUT) argument of type INTEGER(ip_), that should be set to 0 if the routine has been able to evaluate the gradient of the objective function and to a non-zero value if the evaluation has not been possible.

X is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{x}$.

userdata is a scalar INTENT(INOUT) argument of type GALAHAD_userdata_type whose components may be used to communicate user-supplied data to and from the subroutines eval_F, eval_G, eval_H, eval_HPROD and eval_PREC (see Section 2.3.7).

G is a rank-one INTENT(OUT) argument of type REAL(rp_), whose components should be set to the values of the gradient of the objective function $\nabla_x f(\mathbf{x})$ evaluated at the vector $\mathbf{x}$ input in X.

2.5.3 Hessian values via internal evaluation

If the argument eval_H is present when calling GALAHAD_BGO_solve, the user is expected to provide a subroutine of that name to evaluate the values of the Hessian of the objective function $\nabla_{xx} f(\mathbf{x})$. The routine must be specified as

```
SUBROUTINE eval_H( status, X, userdata, Hval )
```

whose arguments are as follows:

status is a scalar INTENT(OUT) argument of type INTEGER(ip_), that should be set to 0 if the routine has been able to evaluate the Hessian of the objective function and to a non-zero value if the evaluation has not been possible.

X is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{x}$.

userdata is a scalar INTENT(INOUT) argument of type GALAHAD_userdata_type whose components may be used to communicate user-supplied data to and from the subroutines eval_F, eval_G, eval_H, eval_HPROD and eval_PREC (see Section 2.3.7).

Hval is a scalar INTENT(OUT) argument of type REAL(rp_), whose components should be set to the values of the Hessian of the objective function $\nabla_{xx} f(\mathbf{x})$ evaluated at the vector $\mathbf{x}$ input in X. The values should be input in the same order as that in which the array indices were given in nlp%H.

2.5.4 Hessian-vector products via internal evaluation

If the argument eval_HPROD is present when calling GALAHAD_BGO_solve, the user is expected to provide a subroutine of that name to evaluate the sum $\mathbf{u} + \nabla_{xx} f(\mathbf{x})\mathbf{v}$ involving the product of the Hessian of the objective function $\nabla_{xx} f(\mathbf{x})$ with a given vector $\mathbf{v}$. The routine must be specified as

```
SUBROUTINE eval_HPROD( status, X, userdata, U, V, got_h )
```

whose arguments are as follows:

status is a scalar INTENT(OUT) argument of type INTEGER(ip_), that should be set to 0 if the routine has been able to evaluate the sum $\mathbf{u} + \nabla_{xx} f(\mathbf{x})\mathbf{v}$ and to a non-zero value if the evaluation has not been possible.

X is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{x}$.

userdata is a scalar INTENT(INOUT) argument of type GALAHAD_userdata_type whose components may be used to communicate user-supplied data to and from the subroutines eval_F, eval_G, eval_H, eval_HPROD and eval_PREC (see Section 2.3.7).

U is a rank-one INTENT(INOUT) array argument of type REAL(rp_) whose components on input contain the vector $\mathbf{u}$ and on output the sum $\mathbf{u} + \nabla_{xx} f(\mathbf{x})\mathbf{v}$.

V is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{v}$.

got_h is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If the Hessian has already been evaluated at the current $\mathbf{x}$ got_h will be PRESENT and set .TRUE.; if this is the first time the Hessian is to be accessed at $\mathbf{x}$, either got_h will be absent or PRESENT and set .FALSE.. This gives the user the opportunity to reuse "start-up" computations required for the first instance of $\mathbf{x}$ to speed up subsequent products.

2.5.5 Hessian-sparse-vector products via internal evaluation

If the argument eval_SHPROD is present when calling GALAHAD_BGO_solve, the user is expected to provide a subroutine of that name to evaluate the product $\mathbf{u} = \nabla_{xx} f(\mathbf{x})\mathbf{v}$ involving the Hessian of the objective function $\nabla_{xx} f(\mathbf{x})$ and a given sparse vector $\mathbf{v}$, and to return the nonzero components of the result $\mathbf{u}$. This routine is **not required** if the user has set control%hessian_available to .TRUE. and has made the values of $\nabla_{xx} f(\mathbf{x})$ available either by calls to eval_H (see §2.5.3) or by reverse communication (see §2.6). If needed, the routine must be specified as

```
SUBROUTINE eval_SHPROD( status, X, userdata, nnz_v, INDEX_nz_v, V,      &
                        nnz_u, INDEX_nz_u, U, got_h )
```

whose arguments are as follows:

status is a scalar INTENT(OUT) argument of type INTEGER(ip_), that should be set to 0 if the routine has been able to evaluate the sum $\mathbf{u} + \nabla_{xx} f(\mathbf{x})\mathbf{v}$ and to a non-zero value if the evaluation has not been possible.

X is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{x}$.

userdata is a scalar INTENT(INOUT) argument of type GALAHAD_userdata_type whose components may be used to communicate user-supplied data to and from the subroutines eval_F, eval_G, eval_H, eval_HPROD and eval_PREC (see Section 2.3.7).

nnz_v is a scalar INTENT(IN) argument of type INTEGER(ip_), that specifies the number of nonzeros in the input sparse vector $\mathbf{v}$.

INDEX_nz_v is a rank-one INTENT(IN) array argument of length at least nnz_v and type INTEGER(ip_) whose first nnz_v components give the indices of the nonzero components of $\mathbf{v}$.

V is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components INDEX_nz_v(i), i $= 1,\ldots,$ nnz_v, hold the nonzero values of $\mathbf{v}$. Any other components should be ignored.

nnz_u is a scalar INTENT(OUT) argument of type INTEGER(ip_), that gives the number of nonzeros in the output vector $\mathbf{u}$.

INDEX_nz_u is a rank-one INTENT(OUT) array argument of length at least nnz_u and type INTEGER(ip_) whose first nnz_u components give the indices of the nonzero components of the computed product $\mathbf{u}$.

U is a rank-one INTENT(OUT) array argument of type REAL(rp_) whose components INDEX_nz_u(i), i $= 1,\ldots,$ nnz_u, hold the nonzero values of $\mathbf{u}$. The remaining components should be ignored.

got_h is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If the Hessian has already been evaluated at the current $\mathbf{x}$ got_h will be PRESENT and set .TRUE.; if this is the first time the Hessian is to be accessed at $\mathbf{x}$, either got_h will be absent or PRESENT and set .FALSE.. This gives the user the opportunity to reuse "start-up" computations required for the first instance of $\mathbf{x}$ to speed up subsequent products.

2.5.6 Preconditioner-vector products via internal evaluation

If the argument eval_PREC is present when calling GALAHAD_BGO_solve, the user is expected to provide a subroutine of that name to evaluate the product $\mathbf{u} = \mathbf{P}(\mathbf{x})\mathbf{v}$ involving the user's preconditioner $\mathbf{P}(\mathbf{x})$ with a given vector $\mathbf{v}$. The symmetric matrix $\mathbf{P}(\mathbf{x})$ should ideally be chosen so that the eigenvalues of $\mathbf{P}(\mathbf{x})(\nabla_{xx} f(\mathbf{x}))^{-1}$ are clustered. This subroutine will **only be required** if control%norm = -3, and the user prefers a subroutine call to that provided by reverse communication with inform%status = 6 (see §2.6). The routine must be specified as

```
SUBROUTINE eval_PREC( status, X, userdata, U, V )
```

whose arguments are as follows:

status is a scalar INTENT(OUT) argument of type INTEGER(ip_), that should be set to 0 if the routine has been able to evaluate the product $\mathbf{P}(\mathbf{x})\mathbf{v}$ and to a non-zero value if the evaluation has not been possible.

X is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{x}$.

userdata is a scalar INTENT(INOUT) argument of type GALAHAD_userdata_type whose components may be used to communicate user-supplied data to and from the subroutines eval_F, eval_G, eval_H and eval_PREC (see Section 2.3.7).

U is a rank-one INTENT(OUT) array argument of type REAL(rp_) whose components on output should contain the product sum $\mathbf{u} = \mathbf{P}(\mathbf{x})\mathbf{v}$.

V is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{v}$.

2.6 Reverse Communication Information

A positive value of inform%status on exit from BGO_solve indicates that GALAHAD_BGO_solve is seeking further information—this will happen if the user has chosen not to evaluate function or derivative values internally (see Section 2.5). The user should compute the required information and re-enter GALAHAD_BGO_solve with inform%status and all other arguments (except those specifically mentioned below) unchanged.

Possible values of inform%status and the information required are

2. The user should compute the objective function value $f(\mathbf{x})$ at the point $\mathbf{x}$ indicated in nlp%X. The required value should be set in nlp%f, and data%eval_status should be set to 0. If the user is unable to evaluate $f(\mathbf{x})$—for instance, if the function is undefined at $\mathbf{x}$—the user need not set nlp%f, but should then set data%eval_status to a non-zero value.

3. The user should compute the gradient of the objective function $\nabla_x f(\mathbf{x})$ at the point $\mathbf{x}$ indicated in nlp%X. The value of the i-th component of the gradient should be set in nlp%G(i), for $i = 1,\ldots,n$ and data%eval_status should be set to 0. If the user is unable to evaluate a component of $\nabla_x f(\mathbf{x})$—for instance, if a component of the gradient is undefined at $\mathbf{x}$—the user need not set nlp%G, but should then set data%eval_status to a non-zero value.

4. The user should compute the Hessian of the objective function $\nabla_{xx} f(\mathbf{x})$ at the point $\mathbf{x}$ indicated in nlp%X. The value l-th component of the Hessian stored according to the scheme input in the remainder of nlp%H (see Section 2.3.2) should be set in nlp%H%val(l), for $l = 1,\ldots,$ nlp%H%ne and data%eval_status should be set to 0. If the user is unable to evaluate a component of $\nabla_{xx} f(\mathbf{x})$—for instance, if a component of the Hessian is undefined at $\mathbf{x}$—the user need not set nlp%H%val, but should then set data%eval_status to a non-zero value.

5. The user should compute the product $\nabla_{xx} f(\mathbf{x})\mathbf{v}$ of the Hessian of the objective function $\nabla_{xx} f(\mathbf{x})$ at the point $\mathbf{x}$ indicated in nlp%X with the vector $\mathbf{v}$ and add the result to the vector $\mathbf{u}$. The vectors $\mathbf{u}$ and $\mathbf{v}$ are given in data%U and data%V respectively, the resulting vector $\mathbf{u} + \nabla_{xx} f(\mathbf{x})\mathbf{v}$ should be set in data%U and data%eval_status should be set to 0. If the user is unable to evaluate the product—for instance, if a component of the Hessian is undefined at $\mathbf{x}$—the user need not set nlp%H%val, but should then set data%eval_status to a non-zero value.

6. The user should compute the product $\mathbf{u} = \mathbf{P}(\mathbf{x})\mathbf{v}$ of their preconditioner $\mathbf{P}(\mathbf{x})$ at the point $\mathbf{x}$ indicated in nlp%X with the vector $\mathbf{v}$. The vectors $\mathbf{v}$ is given in data%V, the resulting vector $\mathbf{u} = \mathbf{P}(\mathbf{x})\mathbf{v}$ should be set in data%U and data%eval_status should be set to 0. If the user is unable to evaluate the product—for instance, if a component of the preconditioner is undefined at $\mathbf{x}$—the user need not set data%U, but should then set data%eval_status to a non-zero value.

   This value **can only occur** if the user has set control%norm = -3, and has not provided an optional subroutine eval_PREC (see §2.5.6) to compute the required product with the preconditioner.

7. The user should compute the product $\mathbf{h} = \nabla_{xx} f(\mathbf{x})\mathbf{p}$ of the Hessian of the objective function $\nabla_{xx} f(\mathbf{x})$ at the point $\mathbf{x}$ indicated in nlp%X with the sparse vector $\mathbf{p}$. The nonzeros of $\mathbf{p}$ are stored in data%P(data%INDEX_nz_p(data%nnz_p_l:data%nnz_p_u)) while the nonzeros of $\mathbf{h}$ should be returned in data%HP(data%INDEX_nz_hp(1:data%nnz_hp)); the user must set data%nnz_hp and data%INDEX_nz_hp accordingly, and data%eval_status should be set to 0. If the user is unable to evaluate the product—for instance, if a component of the Hessian is undefined at $\mathbf{x}$—the user need not set data%HP, data%INDEX_nz_hp and data%nnz_hp but should then set data%eval_status to a non-zero value.

   This value **will not occur** if the user has set control%hessian_available to .TRUE. and can provide values of $\nabla_{xx} f(\mathbf{x})$ either by calls to eval_H (see §2.5.3) or by reverse communication (see inform%status = 4, above).

23. The user should follow the instructions for 2 **and** 3 above before returning.

25. The user should follow the instructions for 2 **and** 5 above before returning.

35. The user should follow the instructions for 3 **and** 5 above before returning.

235. The user should follow the instructions for 2, 3 **and** 5 above before returning.

2.7 Warning and error messages

A negative value of inform%status on exit from BGO_solve or BGO_terminate indicates that an error has occurred. No further calls should be made until the error has been corrected. Possible values are:

-1. An allocation error occurred. A message indicating the offending array is written on unit control%error, and the returned allocation status and a string containing the name of the offending array are held in inform%alloc_status and inform%bad_alloc, respectively.

-2. A deallocation error occurred. A message indicating the offending array is written on unit control%error and the returned allocation status and a string containing the name of the offending array are held in inform%alloc_status and inform%bad_alloc, respectively.

-3. The restriction nlp%n > 0 or requirement that nlp%H_type contains its relevant string 'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS' or 'DIAGONAL' has been violated.

-4. The bound constraints are inconsistent.

-7. The objective function appears to be unbounded from below on the feasible set.

-9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component inform%factor_status.

-10. The factorization failed; the return status from the factorization package is given in the component inform%factor_status.

-11. The solution of a set of linear equations using factors from the factorization package failed; the return status from the factorization package is given in the component inform%factor_status.

-16. The problem is so ill-conditioned that further progress is impossible.

-17. The step is too small to make further impact.

-18. Too many iterations have been performed. This may happen if control%trb_control%maxit is too small, but may also be symptomatic of a badly scaled problem.

-19. The elapsed CPU or system clock time limit has been reached. This may happen if either control%cpu_time_limit or control%clock_time_limit is too small, but may also be symptomatic of a badly scaled problem.

-82. The user has forced termination of GALAHAD_BGO_solve by removing the file named control%alive_file from unit control%alive_unit.

2.8 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable control of type BGO_control_type (see Section 2.3.3), by reading an appropriate data specification file using the subroutine BGO_read_specfile. This facility is useful as it allows a user to change BGO control parameters without editing and recompiling programs that call BGO.

A specification file, or specfile, is a data file containing a number of "specification commands". Each command occurs on a separate line, and comprises a "keyword", which is a string (in a close-to-natural language) used to identify a control parameter, and an (optional) "value", which defines the value to be assigned to the given control parameter. All keywords and values are case insensitive, keywords may be preceded by one or more blanks but values must not contain blanks, and each value must be separated from its keyword by at least one blank. Values must not contain more than 30 characters, and each line of the specfile is limited to 80 characters, including the blanks separating keyword and value.

The portion of the specification file used by BGO_read_specfile must start with a "BEGIN BGO" command and end with an "END" command. The syntax of the specfile is thus defined as follows:

```
( .. lines ignored by BGO_read_specfile .. )
  BEGIN BGO
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by BGO_read_specfile .. )
```

where keyword and value are two strings separated by (at least) one blank. The "BEGIN BGO" and "END" delimiter command lines may contain additional (trailing) strings so long as such strings are separated by one or more blanks, so that lines such as

```
BEGIN BGO SPECIFICATION
```

and

```
END BGO SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN BGO" and "END" delimiters, specification commands may occur in any order. Blank lines and lines whose first non-blank character is ! or * are ignored. The content of a line after a ! or * character is also ignored (as is the ! or * character itself). This provides an easy manner to "comment out" some specification commands, or to comment specific values of certain control parameters.

The value of a control parameters may be of three different types, namely integer, logical or real. Integer and real values may be expressed in any relevant Fortran integer and floating-point formats (respectively). Permitted values for logical parameters are "ON", "TRUE", ".TRUE.", "T", "YES", "Y", or "OFF", "NO", "N", "FALSE", ".FALSE." and "F". Empty values are also allowed for logical control parameters, and are interpreted as "TRUE".

The specification file must be open for input when BGO_read_specfile is called, and the associated device number passed to the routine in device (see below). Note that the corresponding file is REWINDed, which makes it possible to combine the specifications for more than one program/routine. For the same reason, the file is not closed by BGO_read_specfile.

2.8.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL BGO_read_specfile( control, device )
```

control is a scalar INTENT(INOUT) argument of type BGO_control_type (see Section 2.3.3). Default values should have already been set, perhaps by calling BGO_initialize. On exit, individual components of control may have been changed according to the commands found in the specfile. Specfile commands and the component (see Section 2.3.3) of control that each affects are given in Table 2.1.

command	component of control	value type
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
maximum-current-random-searches	tt %attempts_max	integer
maximum-number-of-evaluations	tt %max_evals	integer
sampling-strategy	tt %sampling_strategy	integer
hypercube-discretization	tt % hypercube_discretization	integer
alive-device	%alive_unit	integer
infinity-value	%infinity	real
minimum-objective-before-unbounded	%obj_unbounded	real
maximum-cpu-time-limit	%cpu_time_limit	real
maximum-clock-time-limit	%clock_time_limit	real
hessian-available	%hessian_available	logical
random-multistart	%random_multistart	logical
space-critical	%space_critical	logical
deallocate-error-fatal	%deallocate_error_fatal	logical
alive-filename	%alive_file	character

Table 2.1: Specfile commands and associated components of control.

device is a scalar INTENT(IN) argument of type INTEGER(ip_), that must be set to the unit number on which the specfile has been opened. If device is not open, control will not be altered and execution will continue, but an error message will be printed on unit control%error.

2.9 Information printed

If control%print_level is positive, information about the progress of the algorithm will be printed on unit control%out. If control%print_level $= 1$, a single line of output will be produced every time the objective function improves. This will include the number of attempts to improve the objective so far, the values of the objective function and the norm of its gradient, and the number of function and gradient evaluations.

If control%print_level ≥ 2 this output will be increased to provide significant detail of each iteration. Further details concerning the attempted solution of the models may be obtained by increasing control%TRB_control%print_level, and control%UGO_control%print_level.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: BGO_solve calls the GALAHAD packages GALAHAD_CLOCK, GALAHAD_SYMBOLS, GALAHAD_NLPT, GALAHAD_USERDATA, GALAHAD_SPECFILE, GALAHAD_SPACE, GALAHAD_RAND, GALAHAD_NORMS, GALAHAD_UGO, GALAHAD_LHS and GALAHAD_TRB.

Input/output: Output is under control of the arguments `control%error`, `control%out` and `control%print_level`.

Restrictions: `nlp%n` > 0 and `nlp%H_type` $\in$ { `'DENSE'`, `'COORDINATE'`, `'SPARSE_BY_ROWS'`, `'DIAGONAL'` }.

Portability: ISO Fortran 95 + TR 15581 or Fortran 2003. The package is thread-safe.

4 METHOD

A choice of two methods is available. In the first, local-minimization-and-probe, approach, local minimization and univariate global minimization are intermixed. Given a current champion $\mathbf{x}_k^S$, a local minimizer $\mathbf{x}_k$ of $f(\mathbf{x})$ within the feasible box $\mathbf{x}^l \leq \mathbf{x} \leq \mathbf{x}^u$ is found using `GALAHAD_TRB`. Thereafter m random directions $\mathbf{p}$ are generated, and univariate local minimizer of $f(\mathbf{x}_k + \alpha\mathbf{p})$ as a function of the scalar α along each $\mathbf{p}$ within the interval $[\alpha^L, \alpha^u]$, where α^L and α^u are the smallest and largest α for which $\mathbf{x}^l \leq \mathbf{x}_k + \alpha\mathbf{p} \leq \mathbf{x}^u$, is performed using `GALAHAD_UGO`. The point $\mathbf{x}_k + \alpha\mathbf{p}$ that gives the smallest value of f is then selected as the new champion $\mathbf{x}_{k+1}^S$.

The random directions $\mathbf{p}$ are chosen in one of three ways. The simplest is to select the components as

$$p_i = \text{pseudo random} \in \begin{cases} \text{[-1,1]} & \text{if } x_i^l < x_{k,i} < x_i^u \\ \text{[0,1]} & \text{if } x_{k,i} = x_i^l \\ \text{[-1,0]} & \text{if } x_{k,i} = x_i^u \end{cases}$$

for each $1 \leq i \leq n$. An alternative is to pick $\mathbf{p}$ by partitioning each dimension of the feasible "hypercube" box into m equal segments, and then selecting sub-boxes randomly within this hypercube using Latin hypercube sampling via `GALAHAD_LHS`. Each components of $\mathbf{p}$ is then selected in its sub-box, either uniformly or pseudo randomly.

The other, random-multi-start, method provided selects m starting points at random, either componentwise pseudo randomly in the feasible box, or by partitioning each component into m equal segments, assigning each to a sub-box using Latin hypercube sampling, and finally choosing the values either uniformly or pseudo randomly. Local minimizers within the feasible box are then computed by `GALAHAD_TRB`, and the best is assigned as the current champion. This process is then repeated until evaluation limits are achieved.

If $n = 1$, `GALAHAD_UGO` is called directly.

We reiterate that there are no theoretical guarantees unless the sampling is huge, and realistically the success of the methods decreases as the dimension and nonconvexity increase. Thus the methods used should best be viewed as heuristics.

References:

The generic bound-constrained trust-region method is described in detail in

A. R. Conn, N. I. M. Gould and Ph. L. Toint (2000), Trust-region methods. SIAM/MPS Series on Optimization,

the univariate global minimization method employed is an extension of that due to

D. Lera and Ya. D. Sergeyev (2013), "Acceleration of univariate global optimization algorithms working with Lipschitz functions and Lipschitz first derivatives" SIAM J. Optimization Vol. 23, No. 1, pp. 508–529,

while the Latin-hypercube sampling method employed is that of

B. Beachkofski and R. Grandhi (2002). "Improved Distributed Hypercube Sampling", 43rd AIAA structures, structural dynamics, and materials conference, pp. 2002-1274.

5 EXAMPLES OF USE

Suppose we wish to minimize the parametric objective function $f(\mathbf{x}) = (x_1 + x_3 + p)^2 + (x_2 + x_3)^2 + a * \cos(\omega x_1) + x_1 + x_2 + x_3$ when the parameters p, a and ω takes the values 4, 1000 and 10 respectively, and each component of x is required to lie in the interval $[-10, 0.5]$. Starting from the initial guess $\mathbf{x} = 0$, we may use the following code:

```
   PROGRAM GALAHAD_BGO_EXAMPLE  !  GALAHAD 4.0 - 2022-03-07 AT 13:30 GMT
   USE GALAHAD_BGO_double                        ! double precision version
   IMPLICIT NONE
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )     ! set precision
   TYPE ( NLPT_problem_type ):: nlp
   TYPE ( BGO_control_type ) :: control
   TYPE ( BGO_inform_type ) :: inform
   TYPE ( BGO_data_type ) :: data
   TYPE ( GALAHAD_userdata_type ) :: userdata
   EXTERNAL :: FUN, GRAD, HESS, HESSPROD
   INTEGER :: s
   INTEGER, PARAMETER :: n = 3, h_ne = 5
   REAL ( KIND = wp ), PARAMETER :: p = 4.0_wp
   REAL ( KIND = wp ), PARAMETER :: freq = 10.0_wp
   REAL ( KIND = wp ), PARAMETER :: mag = 1000.0_wp
   REAL ( KIND = wp ), PARAMETER :: infinity = 10.0_wp ** 20    ! infinity
! start problem data
   nlp%pname = 'BGOSPEC'                         ! name
   nlp%n = n ; nlp%H%ne = h_ne                   ! dimensions
   ALLOCATE( nlp%X( n ), nlp%G( n ), nlp%X_l( n ), nlp%X_u( n ) )
   nlp%X = 0.0_wp                                ! start from zero
   nlp%X_l = -10.0_wp ; nlp%X_u = 0.5_wp ! search in [-10,1/2]
!  sparse co-ordinate storage format
   CALL SMT_put( nlp%H%type, 'COORDINATE', s )  ! Specify co-ordinate storage
   ALLOCATE( nlp%H%val( h_ne ), nlp%H%row( h_ne ), nlp%H%col( h_ne ) )
   nlp%H_row = (/ 1, 2, 3, 3, 3 /) ! Hessian H
   nlp%H_col = (/ 1, 2, 1, 2, 3 /) ! NB lower triangle
! problem data complete
   ALLOCATE( userdata%real( 3 ) )                ! Allocate space for parameters
   userdata%real( 1 ) = p                        ! Record parameter, p
   userdata%real( 2 ) = freq                     ! Record parameter, freq
   userdata%real( 3 ) = mag                      ! Record parameter, mag
   CALL BGO_initialize( data, control, inform ) ! Initialize control parameters
   control%TRB_control%subproblem_direct = .FALSE.  ! Use an iterative method
   control%attempts_max = 1000
   control%max_evals = 1000
   control%TRB_control%maxit = 10
!   control%print_level = 1
!   control%TRB_control%print_level = 1
!   control%UGO_control%print_level = 1
! Solve the problem
   inform%status = 1                             ! set for initial entry
   CALL BGO_solve( nlp, control, inform, data, userdata, eval_F = FUN,         &
                   eval_G = GRAD, eval_H = HESS, eval_HPROD = HESSPROD )
   IF ( inform%status == 0 ) THEN                ! Successful return
     WRITE( 6, "( ' BGO: ', I0, ' evaluations -', /,                           &
    &     ' Best objective value found =', ES12.4, /,                          &
    &     ' Corresponding solution = ', ( 5ES12.4 ) )" )                       &
     inform%f_eval, inform%obj, nlp%X
   ELSE                                          ! Error returns
     WRITE( 6, "( ' BGO_solve exit status = ', I6 ) " ) inform%status
   END IF
   CALL BGO_terminate( data, control, inform )  ! delete internal workspace
   DEALLOCATE( nlp%X, nlp%G, nlp%H%val, nlp%H%row, nlp%H%col, userdata%real )
   END PROGRAM GALAHAD_BGO_EXAMPLE
```

```
SUBROUTINE FUN( status, X, userdata, f )     ! Objective function
USE GALAHAD_USERDATA_double, ONLY: GALAHAD_userdata_type
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), INTENT( OUT ) :: f
REAL ( KIND = wp ), DIMENSION( : ),INTENT( IN ) :: X
TYPE ( GALAHAD_userdata_type ), INTENT( INOUT ) :: userdata
REAL ( KIND = wp ) :: p, mag, freq
p = userdata%real( 1 ) ; freq = userdata%real( 2 ) ; mag = userdata%real( 3 )
f = ( X( 1 ) + X( 3 ) + p ) ** 2 +                                          &
    ( X( 2 ) + X( 3 ) ) ** 2 + mag * COS( freq * X( 1 ) ) +                 &
      X( 1 ) + X( 2 ) + X( 3 )
status = 0
RETURN
END SUBROUTINE FUN

SUBROUTINE GRAD( status, X, userdata, G )    ! gradient of the objective
USE GALAHAD_USERDATA_double, ONLY: GALAHAD_userdata_type
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: G
TYPE ( GALAHAD_userdata_type ), INTENT( INOUT ) :: userdata
REAL ( KIND = wp ) :: p, mag, freq
p = userdata%real( 1 ) ; freq = userdata%real( 2 ) ; mag = userdata%real( 3 )
G( 1 ) = 2.0_wp * ( X( 1 ) + X( 3 ) + p )                                   &
          - mag * freq * SIN( freq * X( 1 ) ) + 1.0_wp
G( 2 ) = 2.0_wp * ( X( 2 ) + X( 3 ) ) + 1.0_wp
G( 3 ) = 2.0_wp * ( X( 1 ) + X( 3 ) + p ) +                                 &
         2.0_wp * ( X( 2 ) + X( 3 ) ) + 1.0_wp
status = 0
RETURN
END SUBROUTINE GRAD

SUBROUTINE HESS( status, X, userdata, H_Val ) ! Hessian of the objective
USE GALAHAD_USERDATA_double, ONLY: GALAHAD_userdata_type
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: H_val
TYPE ( GALAHAD_userdata_type ), INTENT( INOUT ) :: userdata
REAL ( KIND = wp ) :: mag, freq
freq = userdata%real( 2 ) ; mag = userdata%real( 3 )
H_val( 1 ) = 2.0_wp - mag * freq * freq * COS( freq * X( 1 ) )
H_val( 2 ) = 2.0_wp
H_val( 3 ) = 2.0_wp
H_val( 4 ) = 2.0_wp
H_val( 5 ) = 4.0_wp
status = 0
RETURN
END SUBROUTINE HESS

SUBROUTINE HESSPROD( status, X, userdata, U, V, got_h ) ! Hessian-vector prod
USE GALAHAD_USERDATA_double, ONLY: GALAHAD_userdata_type
```

```
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
   INTEGER, INTENT( OUT ) :: status
   REAL ( KIND = wp ), DIMENSION( : ), INTENT( INOUT ) :: U
   REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X, V
   TYPE ( GALAHAD_userdata_type ), INTENT( INOUT ) :: userdata
   LOGICAL, OPTIONAL, INTENT( IN ) :: got_h
   REAL ( KIND = wp ) :: mag, freq
   freq = userdata%real( 2 ) ; mag = userdata%real( 3 )
   U( 1 ) = U( 1 )                                                    &
            + ( 2.0_wp - mag * freq * freq * COS( freq * X( 1 ) ) ) * V( 1 )   &
            + 2.0_wp * V( 3 )
   U( 2 ) = U( 2 ) + 2.0_wp * ( V( 2 ) + V( 3 ) )
   U( 3 ) = U( 3 ) + 2.0_wp * ( V( 1 ) + V( 2 ) + 2.0_wp * V( 3 ) )
   status = 0
   RETURN
   END SUBROUTINE HESSPROD
```

Notice how the parameters p, a and ω are passed to the function evaluation routines via the `real` component of the derived type `userdata`. The code produces the following output:

```
 BGO: 1001 evaluations -
 Best objective value found = -1.0049E+03
 Corresponding solution =  -4.7124E+00 -8.7550E-01  4.4270E-01
```

If the user prefers to provide function, gradient and Hessian information without calls to specified routines, the following code is appropriate.

```
   PROGRAM GALAHAD_BGO_EXAMPLE2  !  GALAHAD 4.3 - 2024-02-07 AT 07:30 GMT
   USE GALAHAD_BGO_double                        ! double precision version
   IMPLICIT NONE
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )    ! set precision
   TYPE ( NLPT_problem_type ):: nlp
   TYPE ( BGO_control_type ) :: control
   TYPE ( BGO_inform_type ) :: inform
   TYPE ( BGO_data_type ) :: data
   TYPE ( GALAHAD_userdata_type ) :: userdata
   INTEGER :: s
   INTEGER, PARAMETER :: n = 3, h_ne = 5
   REAL ( KIND = wp ), PARAMETER :: p = 4.0_wp
   REAL ( KIND = wp ), PARAMETER :: freq = 10.0_wp
   REAL ( KIND = wp ), PARAMETER :: mag = 1000.0_wp
   REAL ( KIND = wp ), PARAMETER :: infinity = 10.0_wp ** 20    ! infinity
! start problem data
   nlp%pname = 'BGOSPEC'                         ! name
   nlp%n = n ; nlp%H%ne = h_ne                   ! dimensions
   ALLOCATE( nlp%X( n ), nlp%G( n ), nlp%X_l( n ), nlp%X_u( n ) )
   nlp%X = 0.0_wp                                ! start from zero
   nlp%X_l = -10.0_wp ; nlp%X_u = 0.5_wp ! search in [-10,1/2]
!  sparse co-ordinate storage format
   CALL SMT_put( nlp%H%type, 'COORDINATE', s )  ! Specify co-ordinate storage
   ALLOCATE( nlp%H%val( h_ne ), nlp%H%row( h_ne ), nlp%H%col( h_ne ) )
   nlp%H_row = (/ 1, 2, 3, 3, 3 /) ! Hessian H
   nlp%H_col = (/ 1, 2, 1, 2, 3 /) ! NB lower triangle
! problem data complete
   CALL BGO_initialize( data, control, inform ) ! Initialize control parameters
   control%TRB_control%subproblem_direct = .FALSE.  ! Use an iterative method
   control%attempts_max = 1000
```

```
   control%max_evals = 1000
   control%TRB_control%maxit = 10
! Solve the problem
   inform%status = 1                          ! set for initial entry
   DO ! Solve problem using reverse communication
     CALL BGO_solve( nlp, control, inform, data, userdata )
     IF ( inform%status == 0 ) THEN  ! Successful return
         WRITE( 6, "( ' BGO: ', I0, ' evaluations -', /,                    &
        &     ' Best objective value found =', ES12.4, /,                   &
        &     ' Corresponding solution = ', ( 5ES12.4 ) )" )                &
       inform%f_eval, inform%obj, nlp%X
       EXIT
     ELSE IF ( inform%status < 0 ) THEN  ! Error returns
       WRITE( 6, "( ' BGO_solve exit status = ', I6 ) " ) inform%status
       EXIT
     END IF
     IF ( inform%status == 2 .OR. inform%status == 23 .OR.                  &
          inform%status == 25  .OR. inform%status == 235 ) THEN ! evaluate f
       nlp%f = ( nlp%X( 1 ) + nlp%X( 3 ) + p ) ** 2 +                       &
          ( nlp%X( 2 ) + nlp%X( 3 ) ) ** 2 + mag * COS( freq * nlp%X( 1 ) ) +  &
            nlp%X( 1 ) + nlp%X( 2 ) + nlp%X( 3 )
     END IF
     IF ( inform%status == 3 .OR. inform%status == 23 .OR.                  &
          inform%status == 35 .OR. inform%status == 235 ) THEN ! evaluate g
       nlp%G( 1 ) = 2.0_wp * ( nlp%X( 1 ) + nlp%X( 3 ) + p )                &
                      - mag * freq * SIN( freq * nlp%X( 1 ) ) + 1.0_wp
       nlp%G( 2 ) = 2.0_wp * ( nlp%X( 2 ) + nlp%X( 3 ) ) + 1.0_wp
       nlp%G( 3 ) = 2.0_wp * ( nlp%X( 1 ) + nlp%X( 3 ) + p )                &
                      + 2.0_wp * ( nlp%X( 2 ) + nlp%X( 3 ) ) + 1.0_wp
     END IF
     IF ( inform%status == 4 ) THEN ! evaluate H
       nlp%H%val( 1 ) = 2.0_wp - mag * freq * freq * COS( freq * nlp%X( 1 ) )
       nlp%H%val( 2 ) = 2.0_wp
       nlp%H%val( 3 ) = 2.0_wp
       nlp%H%val( 4 ) = 2.0_wp
       nlp%H%val( 5 ) = 4.0_wp
     END IF
     IF ( inform%status == 5 .OR. inform%status == 25 .OR.                  &
          inform%status == 35 .OR. inform%status == 235 ) THEN ! evaluate u = Hv
       data%U( 1 ) = data%U( 1 ) + ( 2.0_wp - mag * freq * freq *           &
              COS( freq * nlp%X( 1 ) ) ) * data%V( 1 ) + 2.0_wp * data%V( 3 )
       data%U( 2 ) = data%U( 2 ) + 2.0_wp * ( data%V( 2 ) + data%V( 3 ) )
       data%U( 3 ) = data%U( 3 ) + 2.0_wp * ( data%V( 1 ) + data%V( 2 )     &
                       + 2.0_wp * data%V( 3 ) )
     END IF
     data%eval_status = 0
   END DO
   CALL BGO_terminate( data, control, inform )  ! delete internal workspace
   DEALLOCATE( nlp%X, nlp%G, nlp%H%val, nlp%H%row, nlp%H%col )
   END PROGRAM GALAHAD_BGO_EXAMPLE2
```

This produces the same output.